Preparación del μClinux para "Software Defined Radio" con BF537

Miguel Rodríguez Caudevilla, José Manuel Pardo Martín, César Benavente Peces, Francisco Javier Ortega González
miguelrc@alumnos.upm.es, jmpardo@diac.upm.es, cbpeces@ics.upm.es, fjortega@diac.upm.es
G.I.R.A. - E.U.I.T.Telecomunicación - U.P.M. - Ctra. Valencia km.7 28031.

***Abstract*—The purpose of this study is to prepare the Blackfin STAMP board with a BF537 core to work as a SDR system (Software Defined Radio System) under a *GNU-Linux* platform. Due to all the documentation and support found in the web and the release under GPL (General Public License), the distribution used to develop SDR is μClinux. This work explains what is needed to build a SDR system in the Blackfin board, the inconveniences found in the actual μClinux distribution and all the workarounds and approaches made to start porting the software needed to the operating system. To make it work, some codes of the *linux* kernel, such as device drivers, had to be reviewed so they could adjust to the application demands. In short, this work explains what has been modified in the μClinux distribution and why such changes are done to make SDR possible in the Blackfin board.**

I. Introducción

En los últimos años, la presencia del procesado digital de señal se ha hecho más notable en los receptores de radiofrecuencia, dando lugar a los sistemas SDR (*Software Defined Radio*). Los motivos son claros: partiendo del coste y de la repetitividad con ajustes mínimos, pasando por la flexibilidad y terminando por la posibilidad de reconfiguración en vuelvo del propio sistema. Esto desemboca en que dependiendo de la aplicación se han de desarrollar distintas soluciones de código que pueden ser intercambiables.

Por otro lado, existe un claro movimiento para dotar de SO (Sistema Operativo) a plataformas μC, μP y DSP, ganándose, de esta forma, las ventajas que proporcionaría dicho SO: conectividad, migración de aplicaciones, gestión de memoria, gestión de acceso a los periféricos, etc.

De esta manera, se pretenden implementar los algoritmos relacionados con SDR bajo μCLinux, en concreto sobre la versión portada a la plataforma Blackfin (Analog Devices).

II. Arquitectura

Centrando la atención en la parte especializada en el procesado de un sistema genérico SDR, la arquitectura utilizada es la mostrada en la figura 1. En dicha figura es clara la utilización de un demodulador IQ, que junto con los filtros previos a la digitalización de la señal, conforma un transformador de Hilbert, convirtiendo la señal de entrada en su equivalente compleja.

Para la conversión A/D se ha elegido un conversor de audio completamente separado del D/A. De esta manera existe la posibilidad de trabajar con frecuencias de muestreo diferentes, lo que implicaría que, de utilizar algún diezmado, podría no ser necesario implementar la interpolación correspondiente antes de enviar las muestras al conversor D/A, lo que evidentemente aliviaría la carga computacional que ha de soportar el sistema.

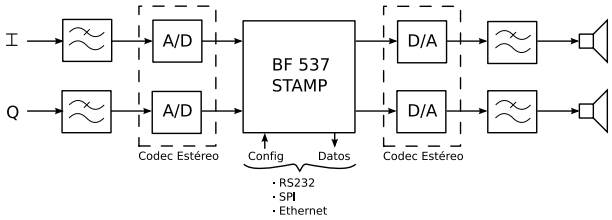


Fig. 1. Diagrama de bloques general para aplicaciones de SDR

Los conversores estéreo de audio elegidos soportan diversos protocolos de comunicaciones serie (LJS, RJS, I^2S). La principal ventaja de este tipo de comunicaciones serie, frente a conversores paralelo, radica en el bajo número de lineas de comunicación necesarias para el trasiego de las muestras. Además, el BF537, posee dos puertos serie capaces de soportar, cada uno, hasta dos conversores estéreo A/D y D/A.

Dentro de los protocolos antes mencionados, el más ampliamente utilizado es el I^2S (*Inter-IC Sound*) ya que es empleado en gran medida en las comunicaciones de audio digitales, muy común para transmitir señales en formato PCM (*Pulse Code Modulation*).

Independientemente del protocolo de comunicaciones, el interfaz físico de los conversores necesita de las siguientes señales, agrupadas según su funcionalidad:

- Una linea de datos dedicada de entrada y otra de salida.
- Reloj de bit (TCLK/RCLK): Genera un pulso por cada bit recibido/transmitido.
- Reloj de trama (TFS/RFS): Indica si se esta recibiendo por el canal izquierdo o derecho. La frecuencia de esta señal coincide con la frecuencia de muestreo.
- Reloj Maestro (MCLK): Se utiliza tanto para gobernar la conversión como para generar las señales de comunicaciones si el conversor funciona como maestro. El conversor D/A siempre deberá ser esclavo, mientras que el conversor A/D, puede ajustarse en función de las necesidades.

Las señales de comunicaciones TFS, TCLK, RFS, RCLK se ajustan perfectamente a las señales del puerto serie que posee el BF537, mientras que para la generación del reloj maestro existen dos posibilidades:

- Generación externa: Reloj dedicado.
- Generación interna: A través del propio BF537.

La primera solución implica la implementación del circuito de reloj correspondiente. Además las señales de comunicación no serán síncronas con dicho reloj maestro si las genera el

BF537, pudiendo acarrear problemas de sincronización. En estas circunstancias, se pueden conseguir mejores resultados si el conversor A/D es maestro de las señales de reloj. El conversor de audio se podría configurar en modo *hardware* o mediante líneas I/O dedicadas ajustando los parámetros acordes con las hojas de especificaciones de los mismos.

Con la segunda opción, todas las señales serían síncronas (ya que derivan del mismo reloj, SCLK, eliminándose la necesidad de implementar la circuitería necesaria y eliminándose los posibles problemas que se pudieran acarrear. Esta aproximación es más flexible ya que permite configurar la comunicación del puerto serie con el conversor mediante *software*. Para utilizar esta opción, es necesario comprobar la posibilidad de que el BF537 genere la señal de reloj MCLK. Se puede llevar a cabo dicho objetivo con la ayuda de uno de los TIMERS (Temporizadores) internos, dividiendo el reloj del sistema, SCLK, y direccionando la salida del TIMER a uno de los puertos de entrada/salida del DSP. Esta es la solución elegida.

III. μCLINUX

La manera de trabajar cambia radicalmente cuando se hace bajo un SO frente a la situación en la que no existe dicho SO.

En aplicaciones solitarias (*standalone*), sin SO, el desarrollador de la misma tiene un total acceso a todos y cada uno de los registro de configuración de los periféricos existentes, así como a la matriz de conmutación que permite seleccionar qué señales estarán presentes en los puertos de salida del DSP (al menos en el caso del BF537), por lo tanto será labor del desarrollador el gestionar correctamente como se han de realizar los accesos a los periféricos, como han de estar configurados y por tanto deberá ser él quien evite las colisiones en los diferentes accesos que puedan surgir durante la ejecución de la aplicación. Es por tanto, una aproximación de elevada flexibilidad en cuanto a lo que se puede llegar a hacer y como conseguirlo y además asegura el carácter determinista de la aplicación.

Por contra en aplicaciones dependientes (*hosted applications*), con un SO, el desarrollador no tiene ningún tipo de acceso a los registros de configuración. Esta es la misión del kernel, que reaccionará en función de las peticiones del usuario. El principal problema que se podría plantear es que el kernel no supiese hacer lo que le pide, o necesita, el usuario. A cambio, el desarrollador de la aplicación sólo deberá centrar su atención en el algoritmo que desea implementar y será el kernel quien se encargue de evitar las colisiones que puedan surgir.

Por tanto, si se trabaja en un entorno con SO basado en *GNU/Linux*, se deberá tener en cuenta cuál es el método usado para acceder a un dispositivo. Si se sigue la tradición de *Linux* en la cuál "todo es un fichero" se puede afirmar que los dispositivos que se pueden gobernar en una computadora serán a su vez un fichero. Un controlador, por lo tanto, es el programa encargado de gestionar las operaciones básicas con ficheros (Las funciones OPEN, CLOSE, IOCTL, READ y WRITE definidas en el estándar ISO/IEC 9899:1990, ANSI C) pero con una ligera diferencia, estos ficheros están asociados a dispositivos y las operaciones que se podrán realizar sobre dichos ficheros quedan determinadas por las características intrínsecas de dicho dispositivo. En la figura 2 (extraída de la documentación web de μClinux para blackfin [1]) se plantea una solicitud de escritura por parte de la aplicación en un dispositivo genérico. Para realizar la escritura el usuario hace una llamada a la función WRITE, esta llamada se redirecciona hacia el controlador que se encargará de preparar el dispositivo para poder llevar a cabo la transferencia del *buffer* del espacio de usuario a la zona de memoria reservada, en espacio de kernel, para dicho dispositivo.

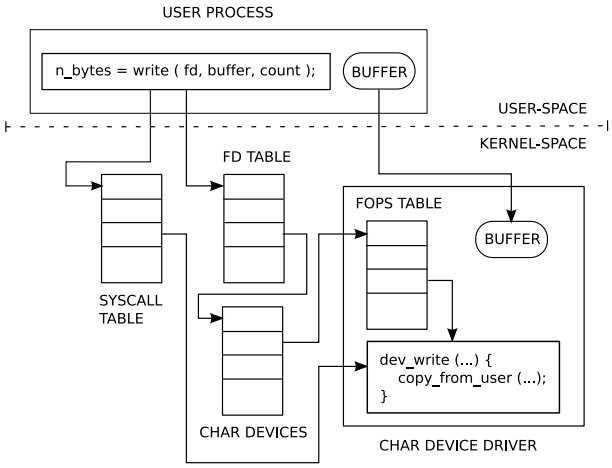


Fig. 2. Ejemplo de escritura en un dispositivo genérico

Puesto que para gobernar a los conversores se necesita acceder a dos periféricos (tal y como se ha planteado el sistema), SPORT y TIMERS, será necesario analizar como están implementados sus respectivos controladores en la distribución de μCLinux, y así comparar con las posibilidades que tendría un desarrollador que trabajase sin SO.

IV. TIMERS

Los TIMERS se caracterizan por su gran versatilidad. Entre las facultades de este dispositivo se encuentran las siguientes características:

- Dividir la señal de SCLK o una señal externa, con la posibilidad de encaminar la señal generada hacia el exterior además de poder generar interrupciones.
- Generar señales PWM, gobernando tanto la frecuencia como el ciclo de trabajo
- Capturar señales PWM para medir su ciclo de trabajo

En el controlador del TIMER que se entrega con la distribución de μClinux no se contemplan todas las posibilidades que el BF537 ofrece para dicho dispositivo [5]. Sólo se permite que el TIMER trabaje como divisor del SCLK, generando interrupción. Además los parámetros de configuración se han de medir en segundos.

Por lo tanto, el TIMER tal y como se encuentra en la distribución de μClinux evidencia una gran falta de configuración para funcionar en un modo distinto al preestablecido además de no encontrarse la posibilidad de trabajar con parámetros de configuración dados en frecuencia, útiles en ciertas situaciones. Entre sus inconvenientes, lo mas grave, surge debido a la generación de interrupciones y la imposibilidad de anular este comportamiento.

Por defecto en los TIMERS se habilitan las interrupciones lo que puede hacer inoperativo el sistema si el usuario

TABLE I
TABLA DE FRECUENCIAS DE MUESTREO POSIBLES

SCLK(MHz)	DIVISIÓN	MCLK(MHz)	MCLK/F_S	F_S(Hz)
100	8	12,5	256	48828
	9	11,1		43402
	10	10,0		39062

configura el TIMER para que la señal tenga una frecuencia elevada. Con una velocidad elevada de interrupciones de prioridad alta y con la consiguiente sobrecarga debido a los cambios de contexto, se pueden provocar retenciones inesperadas en el sistema. Además de esto el TIMER, por defecto, deshabilita la salida física del dispositivo lo que no permite poder encaminar las señales generadas hacia las distintas lineas de salida que ofrece el BF537.

Es evidente que se hace necesario modificar el controlador del TIMER pero por el momento sólo se realizaran los cambios pertinentes para cumplir las necesidades. Se planea modificar, en un futuro cercano, el controlador de manera más profunda, para contemplar todas las posibilidades de configuración.

La frecuencia de muestreo está gobernada, entonces, por el reloj del sistema y por el factor de división del TIMER. Debido a esto, puede que no se consigan las frecuencias indicadas en las hojas de características del A/D y D/A, pero en esencia, lo realmente importante es la relación existente entre el MCLK y TFS. En la tabla I se muestran los valores de TFS que se pueden obtener bajo la distribución de μClinux a partir del reloj SCLK.

V. SPORT

El *hardware* del SPORT del BF537 presenta una gran flexibilidad en cuanto a formatos (protocolos) de comunicaciones se refiere, permitiendo ajustarse a estándares existentes o creando protocolos específicos. Debido a esto, sólo se prestará atención al modo de funcionamiento I^2S, protocolo perfectamente soportado, tanto por los conversores como por el controlador del SPORT existente en la distribución del μCLinux.

En este caso la problemática no viene de las limitaciones en la configuración sino de la forma en la que el controlador recoge las muestras del SPORT y se las entrega a la aplicación que las solicite, en un proceso de lectura. Algo similar ocurre en el proceso de escritura.

Actualmente se contemplan dos modos de transferencia de datos, *PIO* y *DMA a ráfagas*:

- PIO (*Programmed input/output*): La aplicación solicita al controlador un número de muestras (lectura), el controlador recoge dicho número de muestras del SPORT y se las entrega a la aplicación por medio de un bucle *while*. Durante todo este proceso, la aplicación se encuentra bloqueada (Por defecto, en Linux las operaciones de lectura/escritura son bloqueantes). El proceso de escritura se realiza de forma análoga.
- DMA (*Direct Memory Access*) a ráfagas: En este caso, el controlador realiza una petición al kernel para hacer uso de un canal de DMA y se habilita la interrupción asociada a dicho canal, que servirá para notificar la finalización de la transferencia. Cuando la aplicación solicita un conjunto de muestras, el controlador configura el controlador de DMA para que transfiera las muestras desde el SPORT hacia la dirección de memoria indicada por la aplicación, en espacio de usuario, tal y como se muestra en la figura 3. Una vez terminada la configuración se habilita tanto la transferencia DMA como la del SPORT (que hasta ese momento ha estado apagado) quedando el controlador a la espera de que se termine dicha transferencia. Durante todo este proceso, la aplicación también se encuentra bloqueada. El proceso de escritura se realiza de forma análoga.

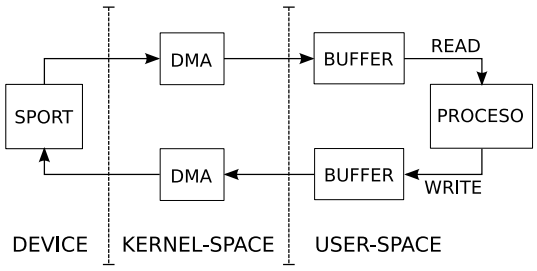
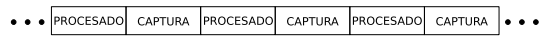


Fig. 3. Diagrama de bloques para *DMA a ráfagas*

El principal problema del modo PIO y DMA radica en el bloqueo que sufre la aplicación durante la adquisición de las muestras.

Este bloqueo incidirá en dos aspectos. Principalmente, si se están leyendo muestras no se puede escribir y viceversa, lo que reduce el tiempo de CPU para la realización de los cálculos del algoritmo de procesado. Por otro lado, mientras que se están leyendo (o escribiendo) las muestras, no se puede realizar el procesado, o lo que es lo mismo, mientras que se está procesando no se pueden capturar muestras, lo que implica una discontinuidad en el flujo de procesado y captura, tal y como se muestra en la figura 4(a)

Para poder solucionar este inconveniente, la situación ideal sería la de poder leer y escribir al mismo tiempo. Estrictamente hablando esto no es posible, pero si el tiempo que se tarda en leer o escribir es muy corto, la situación sería óptima. Además, para poder capturar mientras se están procesando las muestras se propone el diagrama temporal mostrado en la figura 4(b).



(a) Diagrama temporal en modo *DMA a ráfagas*

(b) Diagrama temporal en modo *DMA Autobuffering PinPon*

Fig. 4. Comparativa entre los modos de funcionamiento con DMA habilitado

La forma de alcanzar los objetivos mencionados es con la ayuda del método *DMA autobuffering pin-pon*. Con este método, el canal de DMA se configura en modo 2D (2 Dimensiones) de forma que al terminar el primer *sub-buffer* (o *buffer pin*) comenzaría a llenar el segundo *sub-buffer* (o *buffer pon*), como se puede apreciar en el diagrama de bloques en la figura

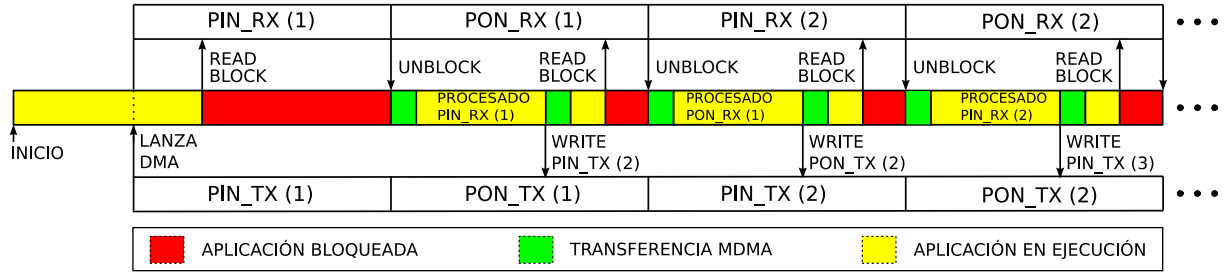


Fig. 5. Diagrama temporal para transferencias en modo *DMA Autobuffering PinPon*

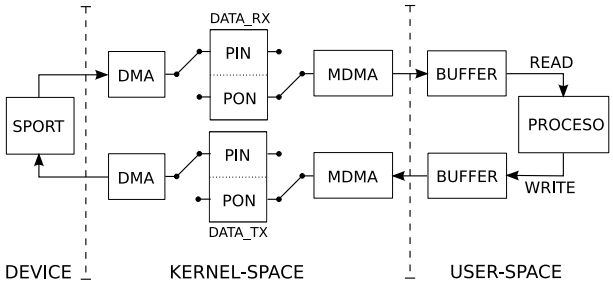


Fig. 6. Diagrama de bloques *DMA Autobuffering Pin-Pon*

6. Para que este ciclo se repita continuamente, sin necesidad de ninguna intervención por parte del kernel, se hace uso también de la capacidad *autobuffering* del controlador de DMA. Mediante esta habilidad del DMA se pueden capturar muestras sin intervención del kernel. Queda claro entones, que con esta estructura la captura se realiza de forma continua.

Otra cuestión importante es decidir desde que *buffer* (pin o pon) se ha de realizar la transferencia hacia la aplicación en un proceso de lectura. Se debe evitar que, en el supuesto de que la aplicación tarde menos tiempo en procesar las muestras que el tiempo necesario para realizar una nueva captura, se transfiera el mismo *buffer* (pin o pon) llevando a cabo la misma captura dos veces. Para realizar esta sincronización, se hace uso de la capacidad del canal DMA de generar interrupciones cada vez que se completa un *sub-buffer*. De esta manera, cuando se realiza una lectura desde la aplicación, esta no se lleva a cabo hasta que el *sub-buffer* haya cambiado (de pin a pon o de pon a pin). Este cambio se puede detectar gracias a la interrupción generada por el DMA al finalizar cada *sub-buffer*, que informa que se ha terminado un *sub-buffer*.

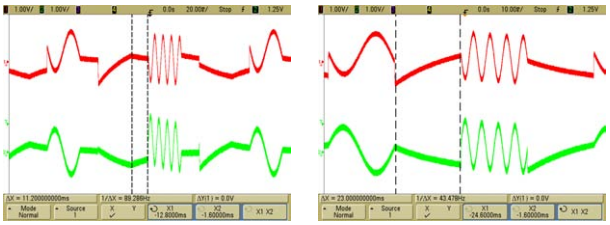
Queda por resolver el mecanismo utilizado para transferir las muestras hacia (lectura) o desde (escritura) la aplicación, en el espacio de usuario. Para esta tarea se utilizará uno de los dos canales de MDMA (Memory DMA) que posee el BF537. El MDMA se encarga de realizar una copia del *sub-buffer* que este libre (considerando como *sub-buffer* libre aquel que el canal de DMA asociado al SPORT no este usando) al *buffer* del espacio de usuario en el momento en el cuál la aplicación realiza la petición de lectura o escritura. De esta manera se consigue que el tiempo de bloqueo tanto en lectura como escritura sea lo más corto posible. En la figura 5 se analizan los diferentes bloqueos que puede sufrir una aplicación genérica.

Este nuevo modo de funcionamiento se ha implementado e integrado en el controlador existente sin detrimento de los otros modos de funcionamiento, pudiéndose elegir y configurar el que más convenga según la aplicación, por medio de la llamada a la función IOCTL.

Actualmente, el controlador implementado, se encuentra en revisión por el grupo de desarrolladores de μCLiunx para Blackfin con el fin de incluirlo en la distribución oficial.

VI. Ejemplos

Con el fin de probar el correcto funcionamiento del controlador desarrollado, se ha implementado una aplicación sencilla que lo único que hace es escribir secuencialmente el contenido de cuatro *buffers* introduciendo un retardo de 10ms entre escrituras. Los *buffers* están compuestos por 1024 muestras, que con una frecuencia de muestreo de 43402 Hz implican un intervalo de 23ms. El contenido de los buffers es: un ciclo sinusoidal, una rampa ascendente, cuatro ciclos sinusoidales y una rampa descendente.



(a) *DMA a ráfagas* (b) *DMA Autobuffering PinPon*

Fig. 7. Comparativa entre escrituras con DMA habilitado

En la figura 7(a) se puede observar el resultado obtenido con el controlador original mientras que en la figura 7(b) se muestra el resultado utilizando el controlador desarrollado. En el primer caso es clara la existencia de tiempos muertos (de aproximadamente 10 ms) entre buffers consecutivos debido al retardo introducido, lo que no ocurre en el segundo caso. De esta forma se asegura una transferencia progresiva y continuada.

Agradecimientos

Este trabajo ha sido llevado a cabo gracias a la financiación del proyecto TEC 2006-08210

References

[1] *http://docs.blackfin.uclinux.org/doku.php*, Versión 1.2, Noviembre 2002: Free Software Foundation, Inc.
[2] J. Corbet, A.Rubini & G. Kroah-Hartman, *Linux Device Drivers*, Tercera Edición, 2005: O'Reilly Media.
[3] Karim Yaghmour, *Building Embedded Linux Systems*, Primera Edición, Abril 2003: O'Reilly Media.
[4] Greg Kroah-Hartman, *Linux Kernel in a Nutshell*, Primera Edición, Diciembre 2006: O'Reilly Media.
[5] *ADSP-BF537 Blackfin Processor Hardware Reference*, Revisión 2.0, Diciembre 2005: Analog Devices, Inc.